

# The Coalgebraic Class Specification Language CCSL —Tutorial—

Hendrik Tews

5th August 2002

**Abstract.** This is a DRAFT VERSION of a tutorial introduction into the Coalgebraic Class Specification language CCSL. The final version will cover coalgebraic specifications, loose and final models, (assertional) refinement, general reasoning about the specification and proofs by coinduction. The running example is a (possibly infinite) coalgebraic stack.

**Acknowledgements.** CCSL and the CCSL compiler have been designed and implemented in the LOOP project on formal methods of object-orientation. All members of the LOOP team contributed to CCSL, often substantially.

## Contents

<b>List of Figures and Tables</b>	<b>3</b>
<b>1. Introduction</b>	<b>4</b>
1.1. History . . . . .	4
<b>2. A Coalgebraic Stack</b>	<b>5</b>
2.1. The Signature . . . . .	6
2.2. The Axioms . . . . .	9
2.3. Running the CCSL Compiler . . . . .	13
<b>3. Developing a Model with PVS</b>	<b>15</b>

---

## List of Figures and Tables

1.	The stack specification in CCSL . . . . .	6
2.	Types in CCSL . . . . .	9
3.	Terms of CCSL I . . . . .	11
4.	Terms of CCSL II . . . . .	12
5.	A simple model for the stack specification . . . . .	16

## 1. Introduction

If you are eager to learn CCSL skip to the next section on page 5. This introduction tells something about the history of CCSL, the underlying mathematics and gives some pointers to related work.

### 1.1. History

The origin of CCSL lays in the seminal paper “An Approach to Object Semantics based on Terminal Co-algebras” by Horst Reichel [?]. In this paper Reichel suggest to use final coalgebras to formalise classes of object-oriented languages. Among others, Bart Jacobs picked this idea up and developed it further in a series of publications [?, ?, ?, ?, ?]. Some examples in these papers are presented in a format that closely resembles todays CCSL.

In 1997 Ulrich Hensel visited Bart Jacobs at the University Nijmegen. Together they realised how one could represent coalgebraic specifications as a shallow embedding in PVS: Signatures (or functors) are represented by (parametric) types, coalgebras are functions, and assertions become predicates. Inheritance of classes can be modelled with nested functors and nested record types.

When Ulrich came back from Nijmegen he suggested to me to write a compiler that translates coalgebraic specifications into this shallow embedding. Such a compiler would awake coalgebraic theory to live in that it would enable us to experiment much more with even larger examples. The idea was fascinating, so we immediately stated coding. I was responsible for the choice of the programming language: Ocaml, which was at version 1.04 back then. Two years later the ICFP Contest judges justified my choice with the statement “Objective CAML is the programming tool of choice for discriminating hackers”.<sup>1</sup>

In November 97 the compiler was able to process simple coalgebraic specifications. Bart Jacobs and Marieke Huisman in Nijmegen contributed mainly to the automatic proof generation. They also worked on details of the embedding into PVS and developed examples. Most of the ocaml code was written in Dresden by Ulrich Hensel (mostly backend) and myself (mostly frontend).

We soon found a name for the cooperation of the two groups in Nijmegen and Dresden and identified ourself as the LOOP-team, developing the LOOP-tool, where LOOP stands for Logic for Object-Oriented Programming.

In the fall of 1997 the people in Nijmegen started to work on a Java frontend for the LOOP-tool. The idea was to parse Java classes into the internal structures of the LOOP-tool and to generate the semantics of the Java input as a shallow embedding in PVS by using the existing backend.

---

<sup>1</sup>See <http://www.cs.virginia.edu/~jks6b/icfp>

---

In Dresden we concentrated on the further development of CCSL. We added data types, support for final semantics and full predicate and relation lifting. Ulrich Hensel implemented most of these features for his PhD thesis. In April 1998 Marieke Huisman presented our first paper about CCSL in Lisbon [?]. Soon afterwards a paper Bart presented a paper about the Java part of the LOOP-tool [?].

From September 98 to March 99 I stayed in Nijmegen. During this visit we started a complete reimplementaion of the LOOP-tool, called version 1. At that time variant types were not supported in the official Ocaml version, but distributed as an Ocaml variant, called Olabl. In the reimplementaion of the LOOP-tool we wanted to exploit the subtyping of both Ocamls class types and the variant types. However, after a number of problems with the Ocaml and the Olabl compilers we had to abandon this approach and also version 1.

The new reimplementaion, version 2, was done with the official Ocaml version. While the reimplementaion of the Java part was soon superior to the old LOOP-tool, the CCSL part made almost no progress for a quite long time. It only started to make progress when Jan Rothe entered the LOOP-team in autumn 1999.

In the beginning the CCSL and the Java part of the LOOP-tool had a lot of overlap. However, later it became clear that it would not make sense to adopt the semantics of Java for a general specification language for object-oriented programs. For example, in Java constructors are propagated during inheritance. For Java this is a sensible decision, because the default initialisation ensures that there are no half-initialised objects. In general however, an inherited constructor cannot initialise an object of a derived class. Therefore, in CCSL constructors are not propagated by inheritance. The increasing separation of CCSL and the Java part in the LOOP-tool lead in February 2001 to their separation into two distinct tools: the CCSL compiler (developed in Dresden) and the LOOP-tool for processing Java with JML annotations (developed in Nijmegen).

## 2. A Coalgebraic Stack

The running example of this tutorial will be a coalgebraic stack specification. That is a specification of a (LIFO-) stack where the stack operations `push`, `top`, and `pop` are specified independently as methods and that also permits infinite stacks.

CCSL specifications are written in ordinary ASCII-files, which you create and edit with your favourite tool.<sup>2</sup> A specification consists of a series of (coalgebraic) class specifications, abstract data type declarations, and ground signature extentions, possibly split over several files (see § ?? for the include directive). In the first part of this tutorial we only consider

---

<sup>2</sup>In case your favourite tool is emacs: The CCSL distribution contains a rudimentary CCSL emacs mode, see the file `INSTALL`.

<b>Begin Stack</b> [ T : Type ] : ClassSpec	
<b>Method</b>	2
push : [Self, T] -> Self;	
top : Self -> lift[T];	4
pop : Self -> Self;	
<b>Constructor</b>	6
new_stack : Self;	
<b>Assertion</b>	8
SelfVar s : Self	
Var t : T	10
top_push : s.push t.top = up t;	
pop_push : pop(push(s,t)) ~ s;	12
<b>Assertion</b>	
SelfVar s : Self	14
empty_pop : bottom?( s.top ) <b>Implies</b> bottom?( s.pop.top );	
<b>Creation</b>	16
top_nil : new_stack.top = bottom;	
<b>End Stack</b>	18

---

Figure 1: The stack specification in CCSL

class specifications. For a complete CCSL grammar see § ??.<sup>3</sup>

## 2.1. The Signature

Figure 1 shows the CCSL source code for the stack specification. It starts with the keyword **Begin** and the name of the specification. It is followed by the declaration of the type parameters in square brackets. For stacks we only need one type parameter  $T$  that stands for the type of elements that can be stored in the stack. The type parameters of CCSL are very similar to PVS: They introduce a form of parametric polymorphism. In general the types they denote can be empty.<sup>4</sup>

---

<sup>3</sup>There are also text and html versions of the grammar in the `doc` directory of the distribution.

<sup>4</sup>If you use CCSL together with Isabelle/Hol then CCSL's semantics is adopted to the universe of nonempty sets of Isabelle/Hol. In this case every type is inhabited.

The type parameters are followed by a colon and the keyword **ClassSpec**. This indicates that **Stack** is a class specification. CCSL also knows abstract data types (keyword **Adt**) and ground signature extensions (keyword **Groundsignature**). The stack specification is closed on line 18 with **End Stack**. The body of the specification in between consists of several sections. Each section starts with a descriptive keyword like **Method** or **Assertion**. In general all sections are optional (as long as the specification is not empty), sections can occur several times and their order is irrelevant (however, for readability I propose to declare the methods before the assertions). The stack specification contains one section for method declarations (**Method**), two sections for method assertions (**Assertion**), one section for constructor declarations (**Constructor**), and one section for constructor assertions or creation conditions (**Creation**).

Class specifications can also contain sections for

**inherit clauses** (keyword **Inherit From**) to inherit from other class specifications

**attributes** (**Attribute**) Attributes are methods of a restricted type. The CCSL compiler generates an update method for every attribute together with some assertions. Attributes support specifications with a record-like state space.

**definitions** (**Defining**) to define some methods in terms of others

**theorems** (**Theorem**) to express properties that are expected to hold. The CCSL compiler turns theorems into proof obligations.

**lifting requests** (**Request**) to request the CCSL compiler to generate behavioural equality for a particular type.

The stack example declares three methods: **push** for pushing one element onto the stack, **top** for inspecting the top value, and **pop** for discarding the top value. The grammar of a method declaration is obvious from the example: The name is followed by a colon, a type expression and a semicolon. The last semicolon in a method section is optional. Before I turn to the types of the three methods I have to make two little excursions.

Like in pure mathematics side effects are not possible in the theorem provers PVS and Isabelle and also in CCSL.<sup>5</sup> That is, there are only constants and no variables: Once a value has been assigned it is impossible to change it. Side effects have to be modelled in the functional way, like you would do in a pure functional programming language, like Haskell [?].

The type expressions contain the special type **Self**. To explain it I have to discuss the formal meaning of the terms *specification* and *model*. Formally, a class specification

---

<sup>5</sup>Imagine you could update the value of  $x$  from 5 to 6. Then from  $x = x$  you could conclude  $5 = 6$ .

describes all possible implementations of a given class interface. A class, that is a particular implementation, fixes a state space (given as the public and private attributes in an object-oriented language) and the functional behaviour of the methods. Formally such an implementation is a model of the specification if it fulfills all the properties that are requested in the specification. An object (or more precisely the state of an object) corresponds to an element in the state space. The special type **Self** stands for this state space, that is, it contains all objects and object states.

The type of `push` is now clear: `push` takes a state and an element of `T` and returns the successor state of the stack.

The type expression for `top` uses the type constructor `lift`. This type constructor is defined in the CCSL prelude, which is hardwired into the CCSL compiler. The type constructor `lift` is typically used to model partial functions (remember that `top` cannot return a result for the empty stack). It disjointly adds the error element `bottom` to its argument. Formally:<sup>6</sup>

$$\text{lift}[T] = \{\text{up}(t) \mid t \in T\} \cup \{\text{bottom}\}$$

There are also some additional functions provided. We will meet them when discussing the assertions on page 10.

The syntax of CCSL's type expressions follows PVS but also permits Isabelle's notion for products (`*`) and function types. The details are in Table 2.

The first argument of all three methods is **Self**. This is actually a requirement. It reflects that in object-oriented programming all methods get the object on which they are invoked as (a usually hidden) first argument. The first argument of type **Self** also ensures that all methods can be combined into one function

$$\mathbf{Self} \longrightarrow \boxed{\dots \mathbf{Self} \dots}$$

Such functions are called coalgebras and the theory of coalgebras provides the basis for the semantics of class specifications in CCSL.

With a coalgebra alone it is not possible to create (or compute) new objects unless one knows already some object. For this reason one can declare **Constructor**'s in CCSL. The type of a constructor must be **Self** or a function type with a codomain of **Self**. If the constructor is a function, **Self** must not occur in its domain. The solely purpose of constructors is to provide (parametrised) initial states. The copy-constructors of C++ that generate a copy of their argument are not constructors in the sense of CCSL.

Our example stack specification declares only one constructor `new_stack` on line 7.

The methods and constructors together form the signature of a specification. The signature deals only with the typing aspects of a specification. It says which operations are

---

<sup>6</sup>The CCSL prelude defines `lift` as a nonrecursive data type, see ??.

- $[T_1, \dots, T_n]$  denotes the  $n$ -fold cartesian product of  $T_1, \dots, T_n$ .
- $[T_1 \rightarrow T_n]$  denotes the function space
- $C[T_1, \dots, T_n]$  is the application of  $T_1, \dots, T_n$  to the type constructor  $C$
- $[T_1, \dots, T_n \rightarrow T]$  is shorthand for  $[[T_1, \dots, T_n] \rightarrow T]$
- The outer brackets around the type in a method declaration can be omitted.<sup>7</sup>
- Predefined types and type constructors: the types **bool**, **Unit** (the one-element type), and **EmptyType**<sup>8</sup>, the special types **Self** and **Carrier**<sup>9,10</sup>, and the type constructors **lift**, **list** (of arity one) and **Coproduct** (of arity two).

For the convenience of Isabelle users there is also the following alternative syntax:

- $T_1 * \dots * T_n$  stands for  $[T_1, \dots, T_n]$
- $T_1 \rightarrow T_2$  abbreviates  $[T_1 \rightarrow T_2]$ . Here the arrow is right associative, that is  $T_1 \rightarrow T_2 \rightarrow T_3$  abbreviates  $[T_1 \rightarrow [T_2 \rightarrow T_3]]$

One can also mix PVS and Isabelle syntax and write  $[T_1, T_2 * T_3 \rightarrow T_4]$  for  $[[T_1, [T_2, T_3]] \rightarrow T_4]$  (note that this is different from  $[[T_1, T_2, T_3] \rightarrow T_4]$ ), but this is not recommended.

---

Figure 2: Types in CCSL

available and which type these operations have. It does not say anything about the behaviour of these operations. For CCSL the signature of a specification determines the notions of bisimulation and invariant for that specification.

## 2.2. The Axioms

The stack example continues on line 8 with the logical properties that are required from the operations in the stack signature. These logical properties are divided into *assertions* and *creation conditions*. Assertions are used to determine the behaviour of the methods. Constructors are not allowed in assertions. The Creation conditions are used to state properties about the constructors. In a creation condition one can use both methods and constructors.

Assertions are formulas that contain one free variable of type **Self**. This way an assertion can be turned into a predicate on the state space. The name of this free variable is given

via the **SelfVar** declaration on line 9. If the **SelfVar** declaration is omitted it defaults to `x`. One can declare other variables like the `t` in line 10. These other variables are implicitly universally quantified. All the variable declarations apply to the entire assertion section. This explains why there are two assertion sections in the stack example: The assertion `empty_pop` should not be universally quantified with `t`.

Any assertion section can contain arbitrary many assertions. Each of them consists of a name, a colon, the actual formula, and a semicolon. Here the semicolon terminates the formula, so it is necessary to have a semicolon after the last assertion. The formula is actually a term of type `bool` because CCSL relies on higher-order logic, which does not distinguish between terms and formulas.

Creation condition have the same syntax as assertions. Except that the **SelfVar** declaration is not permitted.

The syntax of terms and formulas is very similar to PVS with some extensions and some restrictions, see the Figure 3 und 4 for details. The CCSL parser is relatively liberal and accepts different syntactic variants. The assertions in the stack example show some of the possibilities. The assertion `top_push` uses traditional object-oriented dot-notation for method invocation. It also omits the parenthesis' in the function applications. Function application binds stronger than the dot, so the assertion is actually parsed as<sup>11</sup>

$$(((s.push) t).top) = (up t)$$

The `pop_push` assertion uses the methods as ordinary functions. Note that the parenthesis around `push(s, t)` are necessary because function application is left associative.

The assertions use functions and constants that are associated with the type constructor `lift`. There type is as follows:

$$\begin{aligned} \text{bottom} &: \text{lift}[T] \\ \text{up} &: T \Rightarrow \text{lift}[T] \\ \text{bottom?} &: \text{lift}[T] \Rightarrow \mathbf{bool} \\ \text{up?} &: \text{lift}[T] \Rightarrow \mathbf{bool} \\ \text{down} &: \text{lift}[T] \Rightarrow T \end{aligned}$$

The constructors `up` and `bottom` generate all elements of the type `lift[T]`, where `T` stands for an arbitrary instantiation. The recognisers `bottom?` and `up?` decide whether there argument was constructed with `up` or `bottom`. The accessor `down` can be used to access the element of type `T` that is inside an element of `lift[T]` (if it is different from `bottom`).<sup>12</sup>

---

<sup>11</sup>The debug option `-D 1024` advices the CCSL compiler to print all assertion after parsing in a nonambiguous form.

<sup>12</sup>CCSL considers `down` erroneously as function of the above type. Soundness will be recovered by the

**Quantification/Abstraction**

```
Forall( a,b : some_type ) : ...  
Exists( a : some_type, b : some_other_type ) : ...  
Lambda( ... ) : ...
```

One can also use a dot to separate the variable list from the succeeding term.

**Let binding**

```
Let a = ... ;  
    b : some_type = ... ;  
In ...
```

The **let** construct binds sequentially (like in PVS). One can use commas to separate different variable bindings. The last separator before **In** is optional.

**Boolean connectives** There are four binary boolean operators: **Iff**, **Implies**, **Or**, **And**

**Conditional**

```
If ... Then ... Elseif ... Then ... ELSE ...
```

The **Elseif** branch is optional; one can also write **Elsif**. (However the **Else** branch is required.) Note that there is no **Endif**.

**Negation** is denoted by **Not**

**Infix application** CCSL permits all symbolic binary infix operators of PVS, for instance +, #, &, \*, -, /, <, <=. Note that = (Equality) and ~ (behavioural equality) are predefined and cannot be changed.

**Modal operators**

```
Always ... For { method_1, method_2, ... }  
Eventually ... For { method_1, ... }
```

I discuss the modal operators in Section ??

---

Figure 3: Terms of CCSL I

**Case distinction**

```

Cases ... OF
  constructor_1 : ... ;
  constructor_2(arg_1, arg_2) : ...;
Endcases

```

**Function update:** ... **WITH** [ ... := ..., ... := ... ]

**Method invocation** is written as usual, like `x . name`. You can use methods also as ordinary functions and write `name x` for methods that take one argument and `name(x, ...)` for methods with more arguments.

**Function application** can be written without parenthesis, like `fun_term argument`.

**Projections** The fifth projection is `PROJ_5`

**Qualified names:** `some_class [ argument_type, ... ] :: some_name`

**Type annotation:** ( ... : some\_type )

**Tuples:** ( term\_1, term\_2, term\_3 )

---

Figure 4: Terms of CCSL II

Now the meaning of the formulas in the stack specification becomes clear: The assertion `top_push` requires that `top` directly after `push` does not fail *and* yields the element just pushed. Let us consider a stack as empty if the `top` method applied to it fails. With this interpretation the `empty_pop` assertion states that `pop` does not add any elements to an empty stack. The creation condition `top_nil` says that `new_stack` is always empty.

It remains the assertion `pop_push`. It uses behavioural equality (the  $\sim$  relation) to state that the result of `pop` after `push` is indistinguishable from the original stack `s`. Internally `pop(push(s,t))` might be different from `s` but nobody can see the difference from the outside, that is if she or he does only use the three methods `push`, `pop`, and `top`.

Behavioural equality makes it possible that a model of the stack specification contains redundant states or that the states contain redundant information (which is invisible through

---

backend theorem prover. In PVS every use of `down` generates a type-check condition that requires one to prove that the argument of `down` is not `bottom`. Isabelle relies on an universe of nonempty types, there one has `down(bottom) = arbitrary`.

the operations of the signature). Such redundancy is often necessary for certain optimisations. Consider for instance the popular stack implementation via an array and a stack pointer. There `pop` usually only adjusts the stack pointer *but does not* delete the top of the stack. In such an implementation we have `pop(push(s,t)) ≠ s` for almost all  $t$ . Using equality instead of behavioural equality in the `top_push` assertion would exclude these models.

Behavioural equality is an important concept in coalgebraic specification: It replaces equality on types that involve **Self**. Behavioural equality is defined via the notion of bisimulation. In turn the definition of bisimulation is directly derived from the signature of the class specification: A bisimulation is a relation that relates only states that cannot be distinguished by the operations from the signature. For a wide class of signatures there always exists a greatest bisimulation, called bisimilarity.<sup>13</sup>

In the `pop_push` assertion behavioural equality is used directly on terms of type **Self**. In this case behavioural equality coincides with bisimilarity. However, one can also use behavioural equality for compound types. For these types bisimilarity is “lifted” through the type. On types that do not involve **Self** behavioural equality is the same as usual equality. See Section ?? for details.

### 2.3. Running the CCSL Compiler

The CCSL compiler translates CCSL specifications into higher-order logic either for PVS or Isabelle/Hol. The CCSL compiler makes it possible to actually use a CCSL specification. In this tutorial I only discuss how to use CCSL in conjunction with PVS. However, the output generated for Isabelle is almost the same (it is generated from the same internal representation). Only the user interaction of Isabelle is quite different from PVS.

The CCSL compiler is a command line tool in the unix tradition. Assuming that the stack specification is saved in the file `stack.beh` one calls<sup>14</sup>

```
cslc stack.beh
```

from the command line to translate it. Emacs users can of course call the compiler via Emacs’ compile command: `M-x compile`. As result the CCSL compiler generates a number of PVS source files. These files have either suffix `.pvs` and contain PVS specifications or suffix `.prf` for PVS proof scripts. For each specification `spec` in the input the compiler generates a file `spec_basic.pvs` that contains the semantics of `spec` in the logic of PVS. The compiler also generates a lot of standard result and convenience lemmas. For some

---

<sup>13</sup>For the existence of a greatest bisimulation one has to exclude all methods that take arguments of a functional type where **Self** occurs in the type of the function argument. One can allow certain function types for the arguments if one excludes certain infinitary models. See [?, ?] for a precise characterisation.

<sup>14</sup>For Isabelle output use `cslc -isa stack.beh`

of these lemmas there are proof scripts in `spec_basic.prf`. The compiler does also generate the files `ccsl_prelude.pvs` and `ccsl_prelude.prf` that contain those parts of the CCSL prelude that are not predefined in PVS. The generated files can be directly loaded into PVS.

One can place the generated files in a different directory with the switch `-d`. For instance

```
ccslc -d Pvs stack.beh
```

For more compiler switches see the manual page.<sup>15</sup>

Each compiler deletes the old contents of the files it generates (however a backup is made by appending a `~` to the file name). So it is advisable to leave the generated files unchanged. However, this is impossible if one wants to prove some of the lemmas in one of the `_basic.pvs` files. To save the contents of the `_basic.prf` files I use the following makefile rules:

```
prffiles=Stack List_Stack
prf=${$(1)}_basic.prf
save=proof_save_${$(1)}_basic
save:
    for f in $(prffiles) ; do \
        echo cp $(call prf,f) $(call save,f) ; \
        cp $(call prf,f) $(call save,f) ; \
    done
restore:
    for f in $(prffiles) ; do \
        echo cp $(call save,f) $(call prf,f) ; \
        cp $(call save,f) $(call prf,f) ; \
    done
```

The variable `prffiles` holds the names of those files to be saved (without the `_basic`). Then `make save` save a copy and `make restore` restores a version.

The compiler stops on the first error encountered. The compiler is organized in several passes, so it can happen that after fixing one error the next error is reported before the first one. The error messages have the right form for Emacs' error message parser: If the compiler was started by `M-x compile` one can jump to the next error with `M-x next-error`, usually bound to `C-x ``. If transient mark mode is active the part of the input that caused the error is highlighted.

---

<sup>15</sup>The manpage is online available at <http://wwwtcs.inf.tu-dresden.de/~tews/ccsl/ccslc.html>.

---

The error messages are relatively poor. Especially for syntax errors. For syntax errors it is often helpful to consult the CCSL grammar.<sup>16</sup> If this does not help do not hesitate to ask the CCSL mailing list or me.

### 3. Developing a Model with PVS

A model of a specification is a structure that defines the operations that are declared in the signature of the specification such that the assertions of the specification become true. Developing a model can serve two purposes:

1. To prove that the specification is consistent, that is, it contains no logical contradictions and declares no impossible operations.
2. To show that the specification captures the intended behaviour. One can never formally prove the correctness a specification. However, one has often a typical model in mind when writing a specification. Proving that this typical model fulfils the specification increases the confidence in the specification.

A model for a coalgebraic class specification consists of

- a state space
- an interpretation of the methods
- an interpretation of the constructors

*for each* possible interpretation of the type parameters. Of course the interpretations of the methods must fulfill the assertions and interpretations methods and constructors together must fulfill the creation conditions.

To develop such a model in PVS one must place the relevant definitions and proofs into a theory that has the same type parameters as the CCSL specification. It is not permitted to make assumptions on the type parameters via the **Assuming** clause. Further one must not state axioms on the definitions. The theory must contain a proof for the  $\langle \text{spec} \rangle \text{Model?}$  predicate that is generated by the CCSL compiler for the specification  $\langle \text{spec} \rangle$ . These conditions cannot be enforced by PVS. However, it is easy to check them oneself.

Figure 5 shows the PVS source code for a simple, list-based model of the stack specification. We will consider other models of the stack specification, therefore we distinguish this model with the theory name `SimpleStackModel`. The theory has one type parameter `T`, precisely as the stack specification has.

---

<sup>16</sup>The grammar is available online at <http://www.tcs.inf.tu-dresden.de/tews/ccsl/grammar.html>. It is also in `Doc/grammar.html` and `Doc/grammar.txt` in the distribution.

```

SimpleStackModel[T : Type] : Theory
Begin
  SimpleState : Type = list[T]
  Importing StackBasic[SimpleState, T]

  simple_stack : StackSignature[SimpleState, T] = (#
    push := Lambda(x : SimpleState, t : T) : cons(t, x),
    pop := Lambda(x : SimpleState) : Cases x of
      null : null,
      cons(t,y) : y
    EndCases,
    top := Lambda(x : SimpleState) : Cases x of
      null : bottom,
      cons(t,y) : up(t)
    EndCases #)

  simple_new : StackConstructors[SimpleState, T] = (# new_stack := null #)

  x : Var SimpleState
  simple_top_push : Lemma top_push?(simple_stack)(x)
  simple_pop_push : Lemma pop_push?(simple_stack)(x)
  simple_empty_pop : Lemma empty_pop?(simple_stack)(x)
  simple_assert : Lemma StackAssert?(simple_stack)
  simple_create : Lemma StackCreate?(simple_stack)(simple_new)

  simple_model : Proposition StackModel?(simple_stack, simple_new)
End SimpleStackModel

```

---

Figure 5: A simple model for the stack specification

As first (on line 3) it is necessary to define a type for the state space of the model. This is the type that is substituted for **Self** in the specification. This type will typically depend on the type parameters. For the simple model we choose lists over  $T$ . Pushing onto the stack will be consing on the internal list and for the pop method we will take the tail of the list.

As next we import the relevant definitions from the generated stack formalisation. There are two main import points defined by the CCSL compiler. One is the theory **StackBasic**, the other one is called **Stack**. The first one imports the definitions and lemmata for bisim-

---

ulations, bisimilarity, invariants, coalgebra morphisms, and the translated assertions and creation conditions. It is the importing of choice for most tasks. The theory `Stack` contains in addition the definitions for full preicate and relation lifting, an axiomatic model (needed if some other specification uses the type of stacks), and the notion of final stack coalgebras.

The theory `StackBasic` takes an additional first type parameter for the state space of the model.<sup>17</sup> The importing on line 4 uses an exact instantiation because the relevant definition have to be used with this instantiation.

The importing of `StackBasic` makes the record types `StackSignature` and `StackConstructors` available. These types are defined in the (generated) theory `StackInterface`. They capture the signature of the stack specification. The type `StackSignature` is a record that combines all method types. So to define an interpretation for all stack methods on the `SimpleState` and on the type parameter `T` one simply has to define a term of type `StackSignature[SimpleState, T]`. The definition of the methods as elements of the `StackSignature` record contains no surprises: The method `push` conses its argument on the current list; `pop` returns the tail, if the current state is nonempty; and `top` returns the head of the list wrapped by `up` or `bottom` for the empty list.

Almost all the definitions that are generated by the CCSL compiler depend on one, sometimes on two models. However, most of these definitions do not depend on the interpretation of the constructors. Therefore the CCSL compiler generates two independent record types, one for the methods and one for the constructors. The CCSL compiler formalates all definitions as functions. Most of them take as first argument a term of type `StackSignature`.

The type `StackConstructors` combines all constructor types. The CCSL compiler defines the type `<spec>Constructors` only if the specification `<spec>` declares some constructors. Again the definition of the constructors in line 15 is obvious.

To finish the construction of the model we have to prove the last proposition. The function `StackModel?` returns true for two records that contain an interpretation of the methods and one of the constructors, precisely if these interpretations make all assertions and all creation conditions true (for all elements in `SimpleState`). For this simple model one can proof this proposition directly (one grind, an obvious instantiation and another grind will so). However, for pedagogical reasons split this proof into several lemmas.

Lets start the proof at the bottom at the proposition `simple_model`. The definitions of the CCSL compiler are such that most of the times a few cycles of expansion, skolemisation, flattening and case splitting yields the relevant proof obligations. For the proof of

---

<sup>17</sup>Most of the generated theories have this additional type parameter. Some theorie (e. g., for bisimulations or for coalgebra homomorphisms) have two additional type parameters for the (possibly independent) state spaces of two models. The theory that defines full relation lifting has two type parameters for every type parameter of the specification in addition to two state spaces.

`simple_model` we expand `StackModel?` and get the following conjunction:

```
simple_model :
  |-----
  {1} ((StackAssert?(simple_stack)) AND (StackCreate?(simple_stack)(simple_new)))
```

This can of course be proved with the two lemmas `simple_assert` and `simple_create`. Note, that the assert predicate depends only on the interpretation of the methods.

In the proof of `simple_create` the expansion of `StackCreate?` yields

```
simple_create :
  |-----
  {1} ((top_nil?(simple_stack)(simple_new)))
```

The predicate `top_nil?` contains the creation condition `top_nil`. The CCSL compiler generates for each assertion and each creation condition such a predicate. The `Assert?` and `Create?` predicates are always simple conjunctions.

The expansion of `top_nil?` yields

```
|-----
{1} top(simple_stack)(new_stack(simple_new)) = bottom
```

This is almost the original CCSL creation condition. Except that the method invocation is written as function application and that both `top` and `new_stack` have an additional first argument.

Formally the interpretation of the terms in the logic of CCSL depends on the interpretation of the methods and constructors. The CCSL compiler makes this dependency explicit by inserting additional arguments. Methods and constructors are turned into record labels that are applied to the record of all methods or all constructors, respectively. We can see this by expanding `simple_stack`:

```
|-----
{1} CASES new_stack(simple_new) OF null: bottom, cons(t, y): up(t) ENDCASES
    = bottom
```

This is exactly the body of the `top` methods as defined in the simple model with `new_stack(simple_new)` substituted for the argument `x`. Expanding `simple_new` finishes the proof. (Of course `grind` would have proven this lemma without further ado.)

---

In the proof of the lemma `simple_assert` we reach the following state after expanding the `StackAssert?` predicate:

`simple_assert :`

```

|-----
{1} FORALL (x: SimpleState):
    ((top_push?(simple_stack)(x)) AND
     (pop_push?(simple_stack)(x)) AND (empty_pop?(simple_stack)(x)))

```

We can prove this with `skosimp` and the three utility lemmas `simple_top_push`, `simple_pop_push`, and `simple_empty_pop`. Note that the assertions are formalised as predicates on the state space, that is, these predicates take an additional argument `x`. The necessary quantification (that the assertions hold for *all* states) is contained in the `StackAssert?` definition.

Out of the remaining three lemmas only `simple_top_push` is not completely trivial, because it requires to prove that two states are bisimilar. A few expansion and skolemization steps yield the state shown below. Note that `bisim?` has to be expanded twice, because bisimilarity on one model (`bisim?` with one `simple_state` argument) is defined in terms of bisimilarity between two models (`bisim?` with two `simple_state` arguments).

`simple_pop_push :`

```

|-----
{1} EXISTS (R: [[SimpleState, SimpleState] -> bool]):
    ((bisimulation?(simple_stack, simple_stack)(R)) AND
     (R(pop(simple_stack)(push(simple_stack)(x!1, t!1)), x!1)))

```

This requires from us to find a relation (in the form of its characterizing function) that is a bisimulation for the simple stack model. Moreover this relation must relate `pop(simple_stack)(push(simple_stack)(x!1, t!1))` and `x!1`, where `x!1` and `t!1` are arbitrarily chosen values.

In our simple model `pop` after `push` returns the original state. Further, the equality relation is always a bisimulation. Therefore we do `(inst 1 "=")` and reach

```

|-----
{1} ((bisimulation?(simple_stack, simple_stack)(=)) AND
     ((pop(simple_stack)(push(simple_stack)(x!1, t!1)) = x!1)))

```

The second conjunct is trivial. For the first one one could employ the lemma `eq_bisim` that is generated by the CCSL compiler (in theory `StackBisimilarityEqRewrite`). However, `grind` is able to finish the proof.

### 3. *Developing a Model with PVS*

---

We have finished the construction of the simple model with the proof of the utility lemmas, because this completes the proof chain for the proposition `simple_model`. We have thus established that the stack specification is consistent.